Gilgamesh: A Multithreaded Processor-In-Memory Architecture for Petaflops Computing *

Thomas L. Sterling[a] and *Hans P. Zima*[a,b]

[a] NASA Jet Propulsion Laboratory, California Institute of Technology, Pasadena, California, U.S.A.

[b] Institute for Software Science, University of Vienna, Austria

E-mail: tron@cacr.caltech.edu,zima@jpl.nasa.gov

Contents

1 Introduction **2**

2 The MIND PIM Architecture **3**
2.1 MIND Architecture Strategy . . . 3
2.2 MIND Chip Architecture . . . 4
2.3 MIND Node Structure . . . 5
2.4 Optimal MIND Partitioning . . . 6
2.5 Parcels . . . 7
2.6 Virtual Address Handling . . . 8
2.7 Multithreaded Execution . . . 8

3 Elements of the Gilgamesh Execution Model **9**

4 Distributed Collections and the Generation of Data Parallel Code **11**
4.1 Collections . . . 11
4.2 Data Distribution and Alignment . . . 12
4.3 Distribution and Alignment Objects . . . 14
4.4 Runtime Management of Distributed Collections . . . 15
4.5 Putting Things Together: Two Examples . . . 15
4.5.1 Vector Manipulation . . . 15
4.5.2 Sparse Matrix Vector Multiply . . . 17

5 Related Work **18**

6 Conclusions and Future Work **18**

Abstract

Processor-in-Memory (PIM) architectures avoid the von Neumann bottleneck in conventional machines by integrating high-density DRAM and CMOS logic on the same chip. Parallel systems based on this new technology are expected to provide higher scalability, adaptability, robustness, fault tolerance and lower power consumption than current MPPs or commodity clusters. In this paper we describe the design of *Gilgamesh*, a PIM-based massively parallel architecture, and elements of its execution model. Gilgamesh extends existing PIM capabilities by incorporating advanced mechanisms for virtualizing tasks and data and providing adaptive resource management for load balancing and latency tolerance. The Gilgamesh execution model is based on *macroservers*, a middleware layer which supports object-based runtime management of data and threads allowing explicit and dynamic control of locality and load balancing. The paper concludes with a discussion of related research activities and an outlook to future work.

Keywords: Massively parallel systems, PetaFlops computing, resource management, locality, load balancing.

*0-7695-1524-X/02 $17.00 (c) 2002 IEEE

1 Introduction

The integration of COTS (commercial off-the-shelf) processing and memory components to synthesize large high-end computing systems is a compelling and pervasive paradigm either through custom MPPs (e.g. CRI T3E, IBM SP2) or commodity clusters (e.g. Beowulf, NOW), with the largest DOE ASCI and NSF PACI installations of this form. Nonetheless, experience with these systems, which deliver Teraflops scale peak performance, strongly indicates that this approach is severely limited in its efficiency for many important types of application and furthermore that the cost, power, and space requirements make their deployment extremely expensive. While it is expected that these conventional systems will reach beyond 100 Teraflops and possibly 1 Petaflops by the end of the decade, they are unlikely to provide the foundation for future generation architectures delivering cost-effective and efficient sustained performance in the trans-Petaflops regime throughout the next decade.

Although the attraction of exploiting mass-market commodity devices is compelling (some think imperative) the hard empirical evidence strongly indicates the contrary. Measured against peak floating-point performance (admittedly an arguable metric, but nonetheless widely used), many instances of single-digit efficiencies have been observed for real-world applications on some of the largest parallel systems. There are two major reasons for this. First, computing exhibits two distinct classes of execution behavior and contemporary systems do not reflect this in their structure or operation. The first class of behavior relies on high temporal locality of data access, allowing efficient use of caches and registers. Conventional RISC microprocessors do handle this class well when spatial locality is also high. This class is compute intensive meaning the ratio of number of arithmetic operations performed on data to the number of load operations to acquire the operand data is ¿¿ 1. But the second class of behavior involves limited temporal locality with little or no data reuse and is memory intensive. Today's conventional microprocessors handle this class very poorly and a significant part of both the cost and inefficiency of today's systems is due to the ineffective means of performing memory intensive operations. The second reason that COTS technology is a poor building block for scalable parallel computing systems is that their architectures are not derived for this purpose; they are designed for sequential execution or as components of small SMP organizations. They do not have sufficient latency tolerance support and only limited hardware support for overhead tasks related to scalable parallel execution. The conclusion to be drawn is that to make trans-Petaflops scale computing practical and widely available demands an alternate strategy than the integration of components that have been adeveloped for entirely different purposes.

An emerging alternative that may contribute to the solution of this important problem exploits processor-in-memory or PIM technology to provide the foundation for a new class of memory-oriented computing element. PIM is enabled by semiconductor fabrication processes which make possible the co-location of both DRAM (or SRAM) memory cells and CMOS logic devices on the same silicon die. PIM differs from System on a Chip (SoC), in that it connects logic directly to the wide row buffer of the memory stack. PIM provides the opportunity to expose substantially greater memory bandwidth while imposing significantly lower latency and requiring less power consumption than conventional systems. This paper is an early presentation of the architecture and software strategy being developed for a new generation PIM-based high-end computer as part of the Gilgamesh Project conducted at the NASA Jet Propulsion Laboratory and the California Institute of Technology. The *MIND (Memory, Intelligence, and Networking Device)* PIM architecture extends previous PIM capabilities by incorporating advanced mechanisms for virtualizing tasks and data and providing dynamic adaptive resource management mechanisms for load balancing and latency tolerance. MIND chips can be employed in homogeneous configurations such that they provide the only computing resource or in more complex hierarchical structures with conventional or custom architecture external high-speed microprocessors to handle the high locality compute intensive tasks at higher speed.

Developing a software infrastructure for Gilgamesh that combines support for a high level of abstraction with efficient object code generation is a hard problem. Gilgamesh is a massively parallel architecture that provides parallelism at three levels: (1) on-node, (2) across the nodes of a chip, and (3) across chips. Control of locality and load balancing, mapping algorithmic parallelism to the proper level of the architecture, and fault tolerance are key issues to be considered in the design of a Gilgamesh software system. This paper focuses on the first of these topics. Locality plays an important role since off-chip references may incur a significant penalty in terms of latency and bandwidth (while multithreading is expected to hide the latency of off-node but on-chip accesses). Most relevant applications will display irregular and dynamic behavior, making the runtime system a central component of the software architecture. We address this issue by proposing a middleware layer, called the *macroserver* level, which supports object-based runtime manage-

ment of data and threads allowing explicit and dynamic control of locality and load balancing. Specific functionality deals with data parallel processing of large-scale distributed data structures which provides a major source for exploiting massive parallelism. Macroserver support *distributed collections*, a combination of a powerful data structuring concept with a generalized form of distribution and alignment.

This paper is structured as follows. The Gilgamesh architecture and the MIND chip are described in Section 2. Section 3 outlines the salient features of macroservers and their role in the Gilgamesh software architecture, followed by the discussion of a generalized approach to the processing of distributed collections in Section 4. We provide an overview of related work in Section 5 and conclude in Section 6 with some insights into current technical challenges and future work.

2 The MIND PIM Architecture

The goal of the MIND architecture is to provide the first truly general-purpose PIM device with support for virtual tasks and data in a distributed shared memory environment. The objective of the MIND design is to enable performance in the trans-Petaflops regime while significantly enhancing performance-to-cost and power efficiency compared to conventional structures and methods. The MIND architecture is targeted to two classes of system organization, (1) homogenous arrays of interconnected MIND chips providing essentially all processing capabilities, and (2) heterogeneous structures comprising a combination of a MIND-based smart memory subsystem with external high performance microprocessors to balance hardware support for compute intensive and memory intensive processing. The first class of system architectures, the homogenous array of MIND chips, is referred to as Gilgamesh (billions of Logic Gate Assemblies through MESH interconnect) after the NASA sponsored project that initially undertook to develop the MIND architecture. The second class of system architectures include those employing external conventional microprocessors such as that being studied under a current project at Sandia National Laboratory and those incorporating external custom processor architectures such as the DARPA sponsored Cray Inc. Cascade project. This paper focuses on the Gilgamesh class of PIM-based system architecture and on the specific MIND PIM architecture under development at Caltech and NASA Jet Propulsion Laboratory.

2.1 MIND Architecture Strategy

Like other PIM-based systems, the MIND architecture exploits the opportunity for exposing a high degree of memory bandwidth by accessing an entire memory row at a time and by partitioning the total memory capacity of a chip into multiple independently accessible blocks. Through the combined data parallelism of a row-wide access (typically 2048 bits on a conventional DRAM chip) and multiple memory banks on a single MIND chip, peak memory bandwidth available on-chip may exceed by two orders of magnitude that of conventional systems off-chip bandwidth of comparable memory capacity. To take advantage of this available memory bandwidth, a wide ALU is tightly integrated with the row buffer of each memory bank to process row-wide data (under favorable conditions of spatial locality) in parallel to permit maximum memory throughput limited only by the raw DRAM technology cycle time. In addition to the high bandwidth, the tight coupling of wide ALU to row buffer permits low latency data accesses. While the degree of advantage may vary widely depending on the nature, form, and scale of the comparable conventional system, improvements of between a factor of four and ten may not be uncommon with respect to an access request from memory through the network, memory bus, and two levels of cache before a register load is completed. Because many memory oriented operations can be performed on the PIM chip itself, many of these data movements off chip need not occur and the power requirements can be significantly decreased. Also, because many of the memory accesses may be performed exclusively on-chip, the external chip interface bandwidth may be employed to greater effect, reducing contention due to conflicts for remote access.

It should be noted that consistent with Moore's Law combined with expected increases in available memory chip area, memory capacity will continue to expand at approximately a factor of four every three years while DRAM cycle rates are improving at less than 10% per annum. With a moderate increase in microprocessor clock rate, it is projected that by the end of the decade it may take a hundred times as long, measured in clock cycle times, to touch every element on a memory chip as it does today. One of the important advantages of PIM is that it can mitigate this growing imbalance. So far, the aspects of the MIND architecture discussed are shared with other PIM architectures. But MIND, in fulfilling its role as a truly general purpose memory processing device, incorporates a number of advanced and interrelated capabilities not found or proposed together by other PIM architectures. These are now identified.

The MIND architecture manages a virtual name space that is shared among multiple MIND nodes (combination of a memory bank, wide ALU, registers, and control) on a given chip and across an array of MIND chips as well. Support for virtual to physical address translation is included as an intrinsic function of every node and works with a distributed directory table as well as local cached address mappings. Thus a full virtual address space shared across all MIND chips is supported. It is noted that the DIVA project at USC ISI is also developing PIM architecture concepts in support of distributed shared memory.

The MIND architecture employs an efficient light-weight protocol, referred to as parcels, in support of message-driven computation among MIND chips and their memory processor nodes. A parcel is a variable length communication packet that includes both argument values and action specifiers targeted to a designated destination. Simple loads and stores can be performed this way, of course, but so can more complex instantiation of remote tasks. Parcels allow work to be moved to the data as well as more conventionally move data to the work. Under specific conditions, parcels can significantly improve efficiency of bandwidth resource usage and contribute to load balancing. But of equal importance is that parcels are a latency hiding mechanism, supporting decoupled computation. Various forms of parcel computation are being pursued by other PIM projects as well including the DIVA project, the University of Notre Dame's PIM-lite project, and the work at the University of Delaware on percolation. Prior art includes but is not restricted to the MIT J-Machine project.

The MIND architecture incorporates fine grain multithreading context switching mechanisms as an intrinsic component of every memory-processor node on a MIND chip. While ordinarily hardware multithreading is employed as a latency hiding strategy (e.g. MTA), MIND also uses multithreading as a mechanism for unifying management of local resources to simplify processor logic design and to increase utilization of critical resources. It replaces several separate mechanisms ordinarily associated with conventional processor designs. Its hardware support for single-cycle context switching and prioritized threads permits efficient interleaving of operations from independent actions driving several different subcomponents of the node architecture. It also provides rapid response to incident parcels and exceptions as well as real time response to external signals.

The MIND architecture includes some additional mechanisms to detect faults and isolate them. This occurs both on the memory-processor node boundaries and for certain classes of faults within an individual node. While not intended to achieve "nonstop computing", its partial coverage of the fault space enables the MIND architecture to deliver high reliability through graceful degradation rather than to suffer single point failures and catastrophic shut down.

To provide the highest degree of usability, the MIND chip incorporates a parallel interface to connect to conventional memory buses to permit integration with COTS systems such as workstations and servers. In its simplest form, such an arrangement does not permit full and general use of MIND chips but does make available some of the capabilities of PIM to such widely available systems.

This paper will discuss several of these aspect of MIND in more detail while leaving others for future coverage. Specifically, virtual naming, parcel communication, and multithreading which are among the principle contributions of MIND to PIM architecture will be addressed more fully in the following sections.

2.2 MIND Chip Architecture

Here we give a brief overview of the MIND chip, its principle subsystems, and their interrelationships. The organization of the MIND chip (Fig.1) interconnects the memory/processor *nodes* to each other, to shared functional on-chip resources, and to external interfaces by means of an on-chip interconnection communication network. The topology of this network depends on the number of nodes co-resident on the chip and the degree of redundancy required for a given level of reliability (through graceful degradation).

The nodes, discussed in more detail below in Section 2.3, incorporate DRAM memory blocks and the logic and control required to provide access to the stored data and perform operations on them. These nodes provide all of the storage of the chip and most of the information processing capability as well as task execution control. Local wide registers of each node serve multiple roles including thread state, instruction cache, vector register, translation lookaside buffers, parcel buffers, and temporary memory row buffers. The node includes execution control for multiple threads with the state of each thread represented by an assigned wide register. The ALU of the node is wide and capable of performing multiple operations on separate fields simultaneously. The specific width of the ALU depends on the time/space tradeoff optimization performed as a number of register to register operations can be performed by the CMOS ALU in the time of a single full row DRAM access. The ALU instruction set includes basic logic functions, variable length integer operations, byte-level permutations, and dual operations. These last are described in the next section. Flow

control instructions, beside those common to every uniprocessor architecture, include thread management operations and parcel handling instructions.

The external interfaces serve specific needs. The external System Memory Bus Interface (a parallel interface) provides the means for connecting MIND chips to conventional workstation and server motherboard memory buses. This interface permits MIND chips to be treated as dumb memory or, through an address mapped protocol, to support higher functions to be performed by the MIND node logic on behalf of controlling system processor(s). The parcel interfaces (serial-parallel) support channels for active message-driven computation between MIND chips. Among other aspects of system operation, the parcel interfaces permit data management to be performed entirely within the memory system. This is made possible through a parcel communication fabric that interconnects the MIND chips. It is anticipated that there will be multiple parcel ports to each MIND chip and that the parcel handler hardware can be employed directly as network routers. A chip data streaming interface (parallel) is also provided for rapid high bandwidth data movement on or off the chip. This can be used for access to high speed secondary data storage or to remote sensing devices such as real time cameras. There are some additional signalling lines used for hardware reset and system interrupt.

The MIND chip also incorporates pipelined floating point arithmetic units that are shared among the nodes by means of the on-chip communications interconnect. These can perform at approximately one Gigaflops each and are in addition to the floating point functionality that is achievable through multicycle operations within the nodes themselves. justification of the MIND chip architecture.

Diagram - MIND Chip Architecture

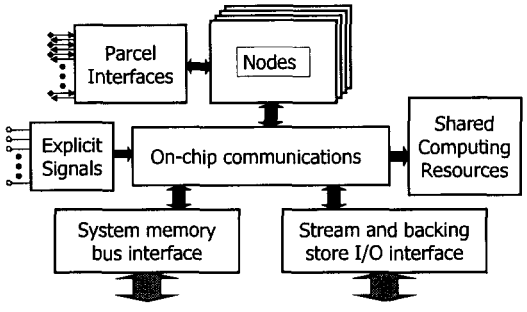

Figure 1: MIND Chip Architecture

2.3 MIND Node Structure

The MIND node provides the core of the MIND operational capability integrating memory blocks with processing logic and control mechanisms. The processing logic of MIND is quite different from a typical core RISC processor, with much simpler control while incorporating very wide ALU logic and registers. The wide ALU is 256 bits wide to match the row segment width delivering DRAM bits every memory subcycle assuming a full row of typically 2048 bits. This is consistent with conventional DRAM desing and operation today. It can support multiple simultaneous SIMD style operations on separate byte fields but also provides operations on selected fields within a wide register delineated by a mask. The MIND ALU permits dual or paired operations on value fields associated with a key or a tag. Depending on a condition to be satisfied by a key the corresponding value field may be modified allowing maximum throughput of the memory bandwidth for sweeping through large data blocks. The ALU does not provide an explicit floating point operation but can perform such operations in a small number of cycles and does so on multiple data pairs simultaneously. With today's technology, a peak floating point throughput of 2 Gigaflops could be achieved per node, comparable to today's microprocessors with much higher integer operation rates depending on operand size. This is justified by modest clock rates of 500 MHz, the width of the ALU, and the number of cycles required to perform a floating point multiply with the MIND node instruction set. Associated with the wide ALU is a permutation network which permits rearrangement of the bytes in a complete row. Many often used permutations can be accomplished in a single cycle, including alignments and one to many distributions.

As will be discussed below, the MIND node architecture incorporates a simple but effective multithreading instruction issue mechanism. A bank of wide registers can be used in a number of ways including as thread state.

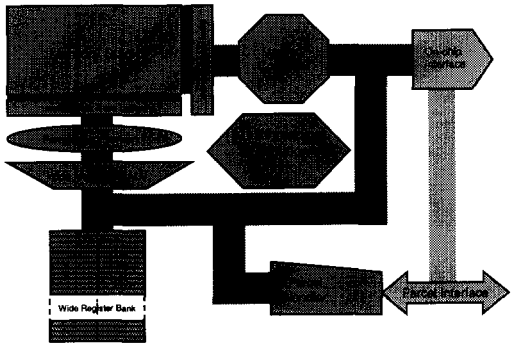

Figure 2: MIND node architecture

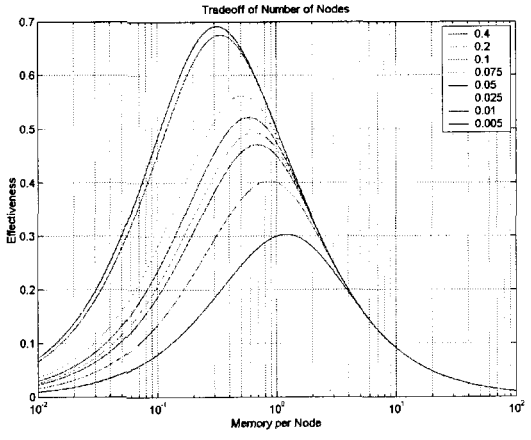

Figure 3: MIND partitioning

2.4 Optimal MIND Partitioning

An important parameter of a MIND chip is the number of memory/processor nodes it contains. For a given DRAM fabrication process, a maximum memory capacity per chip can be assumed. As the number of nodes for a MIND chip increases, the amount of memory capacity per node decreases. In fact, the total memory per chip decreases because of spatial overhead: the amount of die area consumed by the implementation of the processing logic and interfaces per node. For data intensive computation, performance is proportional to sustained memory bandwidth which in turn is related to the concurrency of data access. For a given application exhibiting substantial parallelism, a measure of performance to cost can be reflected by the ratio of the sustained memory bandwidth to the total die area dedicated to the system. Figure 3 shows a series of curves, each one representing a different level of application concurrency. The verticle axis is a normalized measure of this performance to cost metric. It is given as *effectiveness*$=(1-r^s)/(s+1)$, where r is a measure of application memory access concurrency and s is the independent parameter reflecting the memory per node. The horizontal axis is a normalized measure of the memory capacity per node which determines the

number of nodes for fixed user system memory capacity. The unity measure of memory capacity indicates that the space for both the user memory and the processor and interface logic are the same. The curves are derived from a statistical model of memory accesses and suggest that the optimal number of nodes is one for which less than half of the total area is dedicated to memory but well within the first order of magnitude. This is true for a wide range of application parallelism. and its derivation will be presented in the paper.

2.5 Parcels

The parcel is a lightweight active message [32] that permits efficient invocation of remote actions at a node on an external MIND chip. Parcels support message driven computation and permit split or decoupled operations, contributing to system-wide latency tolerance. They are made efficient through hardware support in the MIND architecture.

A parcel is a variable length communication packet that contains sufficient information to perform a remote procedure invocation. The parcel can perform as simple a task as reading a remote virtual addressed location or as complicated as launching an entirely new long term distributed task or move large blocks of data between widely separated MIND chips. The semantic structure of the parcel is the same, even if the actual instantiation of the parcel including its length and contents may vary significantly. All parcels are targeted to a destination or receiver object. Except for low level operations on physical elements (context 0), all parcel destinations are virtual addresses which may identify individual variables, blocks of data, structures, objects, or threads, as well as I/O streams.

The parcel action specifier dictates the operations to be performed at the site of the destination object. The action specifier can be as simple as one of the basic instruction set operations such as load or add, or can identify a method (procedure) to be executed upon receipt of the parcel. A parcel action can also be provided as a string of explicit operations, essentially carrying its own code. The parcel carries argument values with it. These may be used in performing the action or they may be part of a process that carries values on to further locations. An example of the former is the value to be written in to the destination location. An example of the latter is in performing a gather operation across a number of widely distributed points to ultimately be returned to the logical and physical site of execution within the MIND array.

The final element of a parcel is the continuation; that part of the parcel that dictates what is to happen after the action of the parcel is completed. The continuation field could be as simple as a null. Here, all necessary information regarding follow-on work is encoded in the action specifier such as the method. Less trivial but still simple is the return field in the thread register where a remotely read value is to be installed. Often the action of a parcel results in the creation of one or more child parcels. There the continuation dictates that the original parcel be cloned one or more times but sent on to new locations, perhaps carrying some intermediate values. Tree or graph traversal can be carried out this way without repetitious and costly returns to some single thread of control.

Parcels include additional housekeeping fields needed for reliable transport, error detection, routing, and context management. A parcel can vary in length thus providing efficient handling of simple operations with small parcels and effective bandwidth utilization for moving large blocks of data. For example, in one design 64 bit parcels are used to examine and initialize hardware locations, while 256 bit parcels perform basic operations on virtually addressed data and a page of data in a succession of more than a hundred such parcel segments.

The parcel handler is a simple subsystem associated with a node that operates semiautonomously. For some actions, it requires the support of no other subsystem of the node such as when it is engaged in scanning physical data paths and state for purposes of diagnositics, initialization, or reconfiguration. It may work directly with the local memory controller and address translation mechanism without engaging in the thread management and execution units. Finally, the parcel handler can use a parcel as the basis for instantiating a thread which will carry on the intended action. Upon completion of an action, one or more threads may be created and dispatched to remote (off chip) nodes. The parcel handler accepts a return parcel specification from the thread manager and launches it via the inter-MIND chip network. However, for a small subset of simple operations, the parcel handler itself is able to construct a resultant parcel, primarily from the original incident parcel.

2.6 Virtual Address Handling

An important advance of the MIND architecture is the direct handling of a virtual address space for distributed shared memory. No PIM architecture has fully satisfied this requirement, choosing rather (and often

reasonably) to simplify its operations with virtual data by out-sourcing the address translation problem to external conventional processors. But for efficient processing of complex irregular data structures specified by intrinsic meta data including virtual pointers, exploitation of the fine grain data parallelism requires that the MIND architecture node processors be able to accept a virtual pointer and derive a physical location for the specified object. The current memory model is page based (nominally 4K bytes) and allows page placement policy to assign pages to any node of the MIND array. However, favorable placement reduces the time cost of access. The method employed by MIND bears some similarity to that used on the earlier CRI T3E and uses a combination of physical and associative mapping.

The key to the scheme is the distribution of the address directory table. The entry sets making up the directory table for groups of virtual pages are assigned to MIND chips based on their physical names. MIND chips can be grouped such that they comprise an address domain. Any page in that domain will be represented in the directory table segment by means of a hashed table that points to all virtual pages for which it is either responsible or that are guest pages, those not the responsibility of the specific MIND chips in the domain but that are nonetheless stored there. For example, to satisfy some locality constraints, a guest page may be assigned to an MIND chip outside the domain of the guest page's virtual address. Preferential placement of a page is within its domain. Should that be satisfied, a parcel will know through partial direct mapping which set of MIND chips to go to and then will find the exact location of the targeted page. If the target page is a guest page of another domain, then a second hop is required before the parcel finally accesses its intended destination object.

To expedite this process, MIND does not use a conventional fixed size TLB as these do not scale with system size. Rather, MIND exploits its wide registers and wide ALU as a fully associative TLB. More than one register can be used this way simultaneously. Instead of flushing the TLB for a still active context, the contents of the wide register performing the TLB function can be stored in the local main memory and restored to a wide register when the related activity is resumed. Another mechanism to be used is embedding both the virtual and physical address in the pointer of a large distributed data structure. With wide word access being performed in a single memory cycle both fields can be acquired at the same time. Once physical locations for the data structure elements are identified, they are buffered in the data structure itself, thus providing an automatic heuristic for rapid location of remote data.

This technique does not satisfy all requirements for an advanced PIM architecture. For example, it does not allow words of individual pages to be spread out among multiple MIND chips for maximum access bandwidth. It does not resolve conflicts with TLBs of conventional processors that might be integrated in a composite system. And similarly, it does not deal with the problem of cache conflicts associated with those same processors. While MIND does not have caches in the conventional sense, other devices in the system might. It is possible that the invalidate traffic required to maintain consistency may impose an unacceptable overhead burden on the system. Alternatively, disallowing caching in conventional processors of all data that may be manipulated by the MIND chip may have serious consequences for temporal locality. This is an area requiring further work. But the current method is adequate for the PIM-only systems such as Gilgamesh, which is the focus of this paper.

2.7 Multithreaded Execution

The MIND architecture incorporates fine grain multithreading to control the operation of its processor resources. Ordinarily, multithreading had been pursued as a powerful method for hiding system level latency. MIND takes advantage of this capability to hide the relatively short latencies within the chip, but relies on parcels to hide system-wide latency. The value of multithreading to MIND is its uniform and simple mechanism for distributing fine grain actions to functional elements and for avoiding such problems as branch prediction, hazards, and forwarding. It also contributes to keeping the memory blocks as busy as possible to maximize their utilization. Thus multithreading is important to the MIND architecture for improving the efficiency of the system and simplifying its design. It also provides a natural way to provide rapid response to incident parcels with minimum delay.

The multithreaded controller is a simple prioritized round-robin selector-controller. There are no special thread registers. Rather each thread resides in a MIND node wide register with part of the state of the thread stored in specified fields within the designated wide register. When a new thread is created, a wide register is allocated and the multithreaded controller is notified. The controller snoops the bus that loads certain fields within the wide register to determine when the thread is ready to perform its next operation and which resources it will require. The multithreaded hardware controller performs the arbitration between thread requests and resource availability. Requests are classified in terms of priorities. Hard external signals

such as reset get highest priority, context zero parcel requests get the next highest, exceptions the next, memory access the next, and regular operations the lowest priority.

Thread state includes other than the explicit fields of the thread register. A stack frame can be created as part of the kernel heap on the local node memory and pointed to from the thread register. The blocks of instructions are stored temporarily in wide registers and these are also pointed to from the thread register or to the code in memory. Other state including the object to which the thread belongs, or ancillary value registers can also be employed. Threads can share some registers such as instruction registers.

Threads are created, terminated and destroyed, or suspended and stored depending on program and resource requirements. A thread can be created by an incoming parcel or directly by another thread. In both cases, information from one wide register is rearranged in the fields of the new thread register which is assigned by the register supervisor. A thread may be activated as a result of a thread object that acts as a nexus for synchronization. Such an object, which includes the futures construct, determines when the precedence constraints for a thread has been satisfied and activites the thread then. Often this means the arrival of multiple parcels before the thread can begin. If the registers are over subscribed or a thread makes a long response time request, the thread may be suspended. The thread state of the wide register can be efficiently buffered in a single memory access cycle. Termination of a thread is trivial and involves the garbage collection of the thread register by the register supervisor.

Multithreading in PIM was included in a simple form in the J-Machine and was considered more broadly by the HTMT project. The IBM Blue Gene and University of Notre Dame PIM-lite projects are including multithreading in different forms in those architectures. MIND continues this trend and expands the roles of multithreading in PIM design.

3 Elements of the Gilgamesh Execution Model

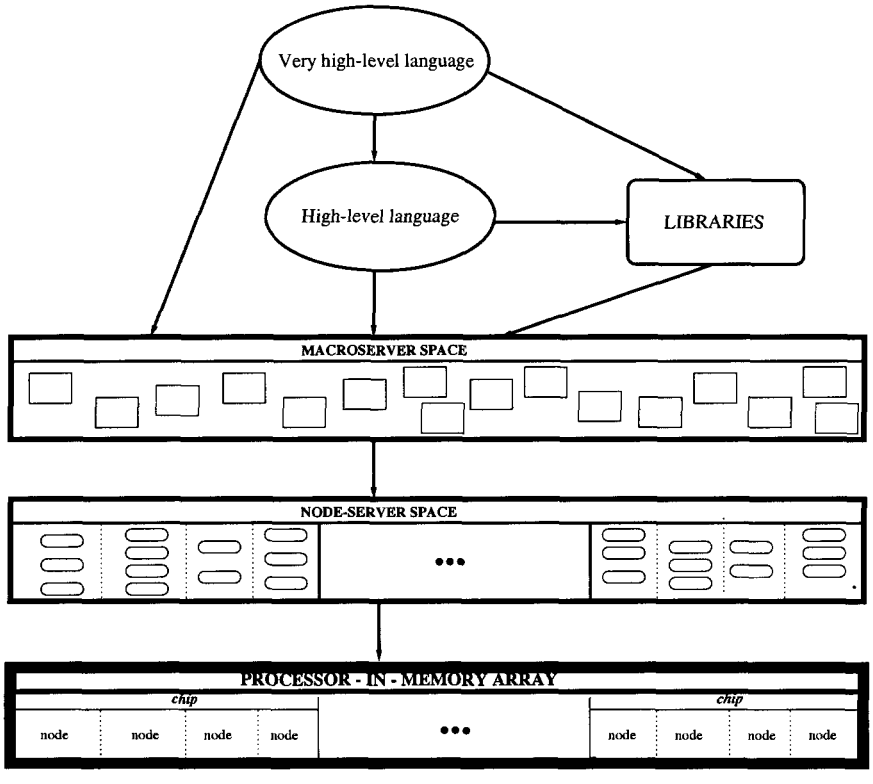

Figure 4: Gilgamesh execution model

Developing a software infrastructure for PIM-based parallel architectures that fulfils the promise of combining high programmer productivity with efficient object code is a hard problem which is currently not yet completely understood, partly because of the limited availablity of such machines. Here are three major issues that have to be resolved:

- **Dealing with the tradeoff between locality and load balancing**
  Gilgamesh is a shared-address space NUMA architecture, and while multithreading is usually able to hide the latency for off-node accesses within a chip, the penalty in terms of bandwidth and latency associated with off-chip references may be severe and must be taken into account when programming the machine. Finding the right tradeoff between locality and load balancing is a difficult problem whose solution must in general be approached heuristically depending on the architecture, the algorithm, and the input data.

- **Exploitation of node parallelism**
  Each node of a MIND chip offers row-wide read/write access to DRAM memory and parallel ALUs operating on wide words. This parallelism must be exploited by the compiler and runtime system if the bandwidth advantage offered by the chip architecture is not to be lost. A related problem is the efficiency of dealing with scalar operands.

- **Fault tolerance**
  PIM-based architectures achieving PetaFlops performance will have chips numbering in the hundred thousands. While multiple nodes on a chip provide a natural redundancy that can be used to achieve fault tolerance on chips, the overall problem of achieving fault tolerance in hardware and software on systems of this size is a matter of on-going research.

This paper addresses the first of these issues. Gilgamesh provides a middleware layer, called the *macroserver* level, which supports object-based runtime management of data and threads allowing explicit and dynamic control of locality and load balancing. Such control is necessary even for many regular applications working on large distributed data aggregates but it is absolutely essential for advanced applications using irregular and dynamic data structures with dynamically varying access patterns, such as adaptive mesh refinement codes operating on semi-structured or unstructured grids. Fig.4 illustrates the role of macroservers in the Gilgamesh software architecture – as a link between high-level languages and very high-level specification systems on the one hand and the low-level software/hardware infrastructure on the other hand –, while Fig.5 provides a view of individual aspects involved in the control of locality and load balancing.

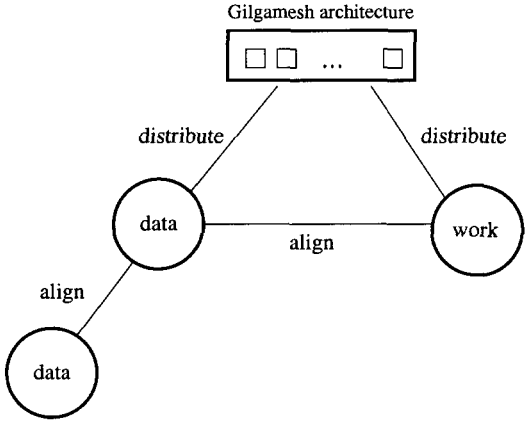

Figure 5: An overview of distribution and alignment control in macroservers

An overview of macroservers has been given in [34, 35]; here we summarize the main ideas underlying this concept. The following Section 4 will then discuss a key aspect – distributed collections – more thoroughly and with a number of fairly detailed examples.

Applications execute as asynchronous networks of macroservers which are dynamic and distributed entities encapsulating variables, methods, threads, and external interfaces much in the sense of object-based computing but with enhanced functionality for performance-oriented computing on PIM-based systems. Macroservers provide a rich semantics for distributing data collections across the nodes of the architecture, including regular and irregular mappings, dynamic redistribution, and a facility for user-defined specialized classes supporting particular access methods and compilation strategies. Data alignment can be used to enforce co-location of elements in different data collections, while sets of threads representing a unit of work (such as a data parallel computation) can be aligned with the distributed collection of data on which they operate (Fig.5).

A complex application may display parallelism at many levels of abstraction. For example, a multidisciplinary optimization for the design of an aircraft [26] will have a coarse-grain layer of heterogeneous task parallelism, involving components for structural analysis, aerodynamics, and optimization with respect to some objective cost function. Within a component, data parallel solvers for large-scale linear equation systems may be used, while at still lower levels, the fine-grain parallelism of vector operations may be exploited. Macroservers provide a number of features for expressing such intricate structures of parallelism and to map them across the levels of parallelism offered by a Gilgamesh system. This includes

- a flexible recursive mechanism for creating threads, by spawning synchronous or asynchronous method activations
- futures [18], which can be tied to threads at the time of creation and provide the means for dynamic status checks, synchronization, and result retrieval after termination. Threads live in user space managed by the macroserver, and are therefore relatively lightweight.
- special semantics for *local methods* which are asserted to operate on a single node. Such methods can exploit the particular hardware functionality associated with nodes such as parallel operations on row-wide data arrangements, and collective operations implemented by the permutation network.

The customers of the macroserver functionality will mainly be high-level language compilers and specification systems: users will be expected to normally program at a higher level of abstraction, possibly in a directive-based extension of Fortran 95 or C/C++, or in an entirely new language. We expect that sophisticated compiler and runtime technology will establish the link from such a high-level API to the macroserver middleware – this is the topic of ongoing and future work. The macroserver functionality will be implemented using *node servers* and their networking via parcels. Node servers are mini objects that perform local functions on a node (see Fig.4). They are very close to the *microservers* advocated by work at the University of Notre Dame [8] and in the DIVA project [16, 17].

4 Distributed Collections and the Generation of Data Parallel Code

SPMD data parallel processing applied to large-scale distributed data structures provides a major source for exploiting massively parallel architectures. Macroserver support focuses on the concept of ***distributed collections***, a combination of a powerful data structuring concept originally introduced in [29] with a generalized form of distribution and alignment as pioneered in languages such as Kali, Vienna Fortran, Fortran D, HPF, and HPC++ [25, 9, 13, 20, 19].

4.1 Collections

Collections, based on work by Sipelstein and Blelloch [29], are homogeneous or heterogeneous aggregates, covering a broad range of methods for structuring, naming, and accessing data. They include multidimensional dense homogeneous arrays in the sense of Fortran or C, LISP list structures, sparse data structures, and sets. In order to be able to express distribution and alignment we assume that any collection, C, is associated with an *index domain* which provides an unambiguous *name* for its primitive elements. For example, the index domain of a Fortran array declared as **REAL** $C(N, M)$ is the set of pairs $\{(i, j) \mid 1 \leq i \leq N, 1 \leq j \leq M\}$. As another example, the elements of tree or list structures can be named using a dot notation as illustrated in Fig. 7. If C denotes a collection, I its index domain, and $i \in I$ is arbitrarily selected, then we let $C(i)$ represent the element of C identified by i.

4.2 Data Distribution and Alignment

Large collections must be distributed across the nodes of the architecture. At the macroserver level, distributions are expressed using a mapping to a parameterized *abstract architecture*, which is understood as a set of *abstract nodes* that communicate with each other via parcels using a communication network. No assumption about the communication topology is made. Abstract nodes are mapped by the low-level system software to the underlying physical architecture; this mapping is transparent and may change dynamically, in particular as a result of component failures. [1]

A data distribution partitions the index domain of a collection into mutually disjoint equivalence classes called distribution segments. They are characterized by two properties: (1) a specific distribution segment defines a scope of locality by forcing all its elements to be mapped to the memory of a single node called its *owner*, and (2) different distribution segments can be mapped to different nodes thus enabling parallel operation across these segments as long as no dependences are violated. Whereas (1) specifies a constraint on the virtual-to-physical mapping, (2) may affect the layout of a collection in virtual memory.

We now describe these concepts in more detail using a simplified formal model. The set of nodes in the abstract architecture is denoted by P. A replication-free *distribution* of a collection C with index domain I is a total function $\delta^C : I \to P$ which specifies for each index, $i \in I$, the abstract node, $\delta^C(i)$, to which $C(i)$ is mapped. For every node $p \in P$ which owns at least one element of C we denote the associated *distribution segment* by $\lambda^C(a) = \{i \in I \mid \delta(i) = a\}$. Note that in order to avoid excessive formalism we use in this paper a simplified model which disallows replication of data across nodes. [2]

The above model does not restrict the structure of the index domain nor the kind of mapping specified by a distribution. As a consequence, distributions can not only express the regular variants of block and cyclic array distributions as defined in HPF-1 [20] and elsewhere, but also arbitrary regular or irregular array mappings including user-defined functions, as well as mappings of more general data aggregates. In many interesting cases, distributions will be computed dynamically, depending on the actual size of collections as well as runtime-specified access patterns (Section 4.3).

An *alignment* establishes a co-location constraint for corresponding elements in two collections. This can be highly useful for enforcing the locality of operations applied to multiple arguments, as discussed later (see Section 4.5.1). The correspondence can be expresssed using an *alignment function*, which establishes a relationship between the index domains of the two collections. Given the distribution of one collection and an alignment function, the distribution of the second collection (the alignee) can be determined. A simple but important case is identity alignment, where all elements of two collections with the same index are mapped to the same node.

We discuss now three examples, covering the distributions of a dense array, a tree data structure, and a sparse matrix. In all cases we assume an abstract architecture with four nodes, $P(1)$ through $P(4)$. While these are toy examples with regard to the sizes of data sets and the abstract architecture involved, they are meant to clarify the basic concepts behind the above definitions as well as their applicability to more general cases.

Example: General Block Array Distribution

Regular block as well as cyclic (round-robin) distributions [20] provide a simple and easily implementable mechanism which is adequate for many codes working on single structured grids. More complex "real" algorithms however, often solve PDEs on semi-structured or unstructured grids. Regular distributions, when applied to such grids, may result in severe load imbalances and large communication overheads since the closeness of data elements in the program representation may not reflect their locality in the underlying physical domain.

General block distributions represent a simple generalization of regular block distributions by allowing a variable block size while retaining the contiguity of the index subdomain associated with each distribution segment. In combination with array element reordering general block distributions can be used to deal with unstructured grids. Fig.6 provides an example for a one-dimensional array $A(1:12)$. The associated distri-

[1] Note that this model distinguishes only local (intra-node) and nonlocal (inter-node) accesses, ignoring references to off-node data on the same chip. This simplification is based on the assumption that multithreading will hide the latency of such accesses.

[2] Replication can play an important role for achieving redundancy in the context of a fault model, or for reducing communication at the cost of additional processor cycles. It is supported by macroservers.

bution segments [3] are given as $\lambda(1) = [1:5]$, $\lambda(2) = [6:7]$, $\lambda(3) = [8:10]$, and $\lambda(4) = [11:12]$. □

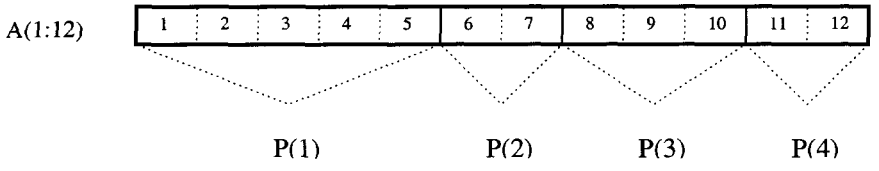

Figure 6: A general block distribution

Example: Distribution of a tree data structure
Fig.7 shows an example for the distribution of a tree data structure. The box style specifies the mapping, yielding the distribution segments $\lambda(1) = \{1, 1.1, 1.1.1, 1.1.1.1\}$, $\lambda(2) = \{1.2, 1.2.1, 1.1.2\}$, $\lambda(3) = \{1.3, 1.3.1\}$, and $\lambda(4) = \{1.3.1.1, 1.3.1.2, 1.3.1.3\}$. □

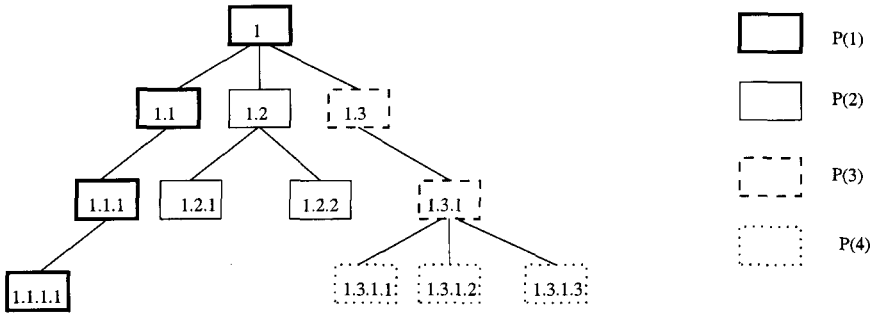

Figure 7: A tree distribution

Example: Sparse Matrix Distribution
Consider a sparse matrix $A(n, m)$ as given in Fig. 8. The elements whose value is 0 are represented by empty boxes; the non-zero elements are explicitly specified and numbered in row major order (for simplicity, we have chosen as the value its position in this numbering scheme). The distribution is irregular, based on the contents of the matrix: the goal is to have (approximately) the same number of nonzero elements in each distribution segment.

In the figure we have $n = 10$ and $m = 8$. The four distribution segments are given as $\lambda(1) = [1:7, 1:5]$ with nonzero elements $(1,2)$, $(3,1)$, $(5,4)$, $(6,5)$; $\lambda(2) = [8:10, 1:4]$ with nonzero elements $(9,2)$, $(9,3)$, $(10,1)$, $(10,4)$; $\lambda(3) = [1:7, 6:8]$ with nonzero elements $(2,7)$, $(3,8)$, $(4,6)$, $(7,7)$; and $\lambda(4) = [8:10, 5:8]$ with nonzero elements $(8,5)$, $(8,8)$, $(9,5)$, $(10,7)$.

A parallel sparse matrix vector multiply algorithm based on this distribution and a special representation of the distribution segments will be discussed in Section 4.5.2. □.

4.3 Distribution and Alignment Objects

In the above discussion distributions were only mentioned in the context of collections to which they were applied: given the index domain, I, of a collection and an abstract architecture, P, we introduced distributions as total mappings, $\delta : I \rightarrow P$. But collections and distributions are orthogonal concepts: we can interpret δ as a ***distribution object*** coming into existence by applying a general mapping rule to I and P as actual arguments. Consider the following example. The cyclic or round robin distribution can be specified in a closed form, using the expression

[3] Nodes are identified by their number only.

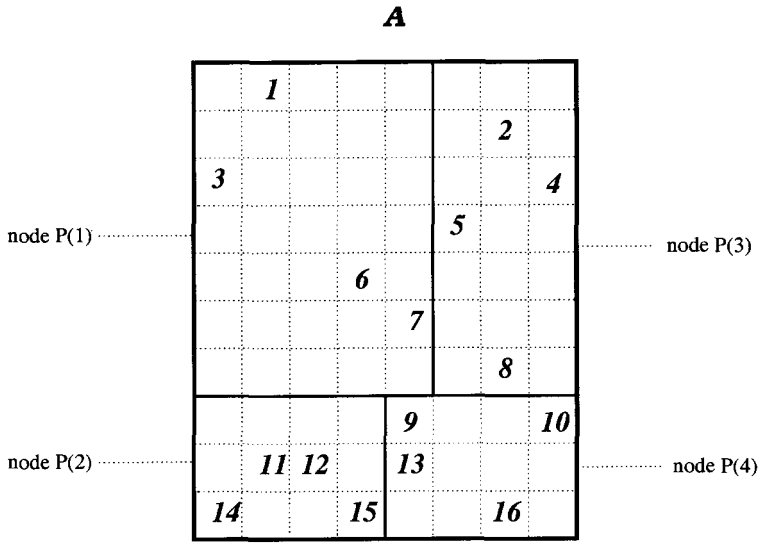

Figure 8: Sparse matrix distribution

$$cyclic(i) = (i\text{-}1) \bmod k + 1 \text{ for all } i,\ 1 \leq i \leq n.$$

Here, k (denoting the number of abstract nodes) and n (for size of an array dimension) are formal parameters, while i is a locally bound variable. This expression can be understood as a template representing the class of all distributions fitting that pattern: combining the template with an arbitrary one-dimensional array index domain of a collection, and an arbitrary abstract node set gives us a specific distribution object as introduced in the previous section. In general, a user may define such a template explicitly in a way similar to that of specifying a procedure [9].

Based on this observation we outline an approach to dealing with data distribution in an object-based framework [9, 36]:

1. *Distribution classes* can be introduced as templates parameterized by two index domains: one associated with a collection and the other with an abstract architecture. The instantiation of such a template with concrete domains yields a *distribution object* that specifies a mapping, δ, as discussed above. A distribution class can be applied to different pairs of index domains, and a given distribution object can be bound to different conforming collections and abstract architectures.

2. A *distributed collection* is a pair (A, δ), where A is a collection and δ is a conforming distribution.

3. Intrinsic methods for distribution objects include those for (1) establishing a binding between collection and a distribution object (the nature of this binding is inherently dynamic), (2) the mapping from an index in the collection domain to a node and a local address on that node, (3) the computation of the distribution segment for a node, and (4) various inquiry functions regarding specific properties of the mapping.

4. Our approach provides functionality for the construction of a library of intrinsic distribution classes. Such templates are characterized by assertions about the collections to which they can be applied and the associated mapping procedure; in addition they may provide specialized auxiliary data structures as well as compilation and runtime methods. Existing libraries may be bound to such a class.

   The discussion of a sparse matrix vector multiply operation in Section 4.5.2 illustrates the idea behind this approach more concretely.

5. Alignment can be understood as a specific distribution constructor function. Examples for other such constructors include incremental redistribution operators as often required for irregularly distributed

collections. In a more general context, alignment objects can be introduced similarly to distribution objects.

4.4 Runtime Management of Distributed Collections

Consider a distributed collection (A, δ) at a given point in time during program execution. As a result of a redistribution which may be triggered by entering a new program phase with modified access patterns (such as in an ADI [24] algorithm or an adaptive mesh refinement code), A will be bound to a new distribution, say δ'. The transition from (A, δ) to (A, δ') is an expensive runtime operation which requires communication between all abstract nodes involved in δ and δ'. We expect that the lightweight parcel mechanism in the Gilgamesh architecture as well as direct hardware support for on-node permutation will make such an operation less expensive and thus more freely usable than on a conventional multiprocessor.

Since redistribution may be performed under an arbitrary control structure, static compiler analysis is generally unable to determine the actual distribution to which a given collection is bound – in fact, a collection may be associated with more than one distribution at a given point of the program. As a consequence, the binding of collections to their distributions is a runtime action, and a representation of the distributed collection must be kept and managed at runtime. *Distribution descriptors* serve that purpose. Ignoring special conventions and assuming for simplicity that we deal with dense arrays, such descriptors contain information about the index domain of the array (this may be a multi-dimensional structure), the index domain of the abstract architecture, and the specification of a distribution function for each distributed dimension. Distribution descriptors are generalizations of layout descriptors as used in HPF compilation systems (see, for example [6]); they may also contain information about access patterns (iterators) and assertions related to optimization. Finally, a distribution descriptor may be used to actually guide the dynamic generation of communication, using its encoded knowledge of the distribution and the collection and architecture index domains.

Looking at the situation from the viewpoint of a distribution object we see that each such object may be associated with more than one conforming array. As a consequence the management of distributed collections can be simplified and memory can be saved by separating the representation of distribution objects from that of collections.

4.5 Putting Things Together: Two Examples

In this section we use two examples to illustrate how some of the concepts introduced above interact and to outline how a data parallel code can be mapped to the Gilgamesh architecture. In the first example, we show how a simple vector code may be mapped to parallel threads operating in nodes of a Gilgamesh abstract architecture. The second example reuses the sparse matrix introduced in Section 4.2 and outlines the generation of a parallel code based on a specialized representation. This code fragment shows, based on an informal HPF-like syntax, the declaration of macroserver classes, their instantiation, the spawning of threads and their linking to futures, and the dynamic establishment of affinity between a set of data parallel threads and a distribution.

4.5.1 Vector Manipulation

Consider a fragment of a vector code consisting of a vector add followed by a sum reduction. The vectors are represented by one-dimensional equal-shaped and identically aligned real arrays A, B, and C, which are general block distributed as shown in Fig.6. The variable S is asumed to be replicated across all nodes.

```
NODES:: P(4)    ! This introduces the abstract node array.
REAL :: A(12), B(12), C(12) DISTRIBUTE(GEN_BLOCK(5,2,3,2)) TO P(1:4)
REAL :: S
...
A = B + C
S = SUM(A)
...
```

This specific code can be fully analyzed at compile time. The specification of the distribution leads to the generation of a distribution object, whose descriptor, d, in a simplified version can be represented as:

$$d = (array\_xdomain = ([1{:}12]),\ node\_xdomain = ([1{:}4]),\ dist = \lambda)$$

Here, λ is given as in Fig.6. The resulting distributed collections are (A, d), (B, d), and (C, d). The execution of this code is initiated by spawning four parallel threads, $T(u)$, $1 \leq u \leq 4$, where $T(u)$ runs on node $P(u)$, operating on the local data stored in this node (Fig.9). This assumes an affinity relationship between the code stored on a node and the data owned by the node. The actual machine-level code will initiate the data parallel operation by broadcasting a parcel parameterized by the node identification to all nodes involved in the operation.

Each thread $T(u)$ executes a parameterized program

```
...
A(λ(u)) = B(λ(u)) + C(λ(u))
S = call (sum_red(A,λ))
...
```

where the first statement is purely local, whereas the sum reduction is a collective operation with communication involving all nodes.

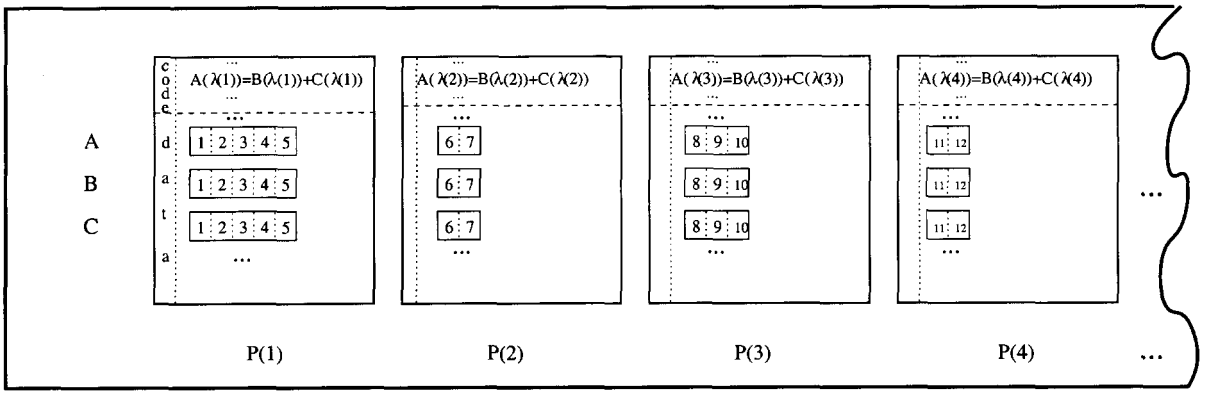

Figure 9: Parallel execution of a vector operation.

We add two remarks. First note that the code executed on each node is parallel in its own right, with the degree of local parallelism depending on the number of array components in the distribution segment. A Gilgamesh node can exploit this parallelism via its capability for loading and storing whole rows of memory, and by using the multiple ALUs on each node to perform component operations in parallel. Moreover, special hardware supports the efficient computation of the partial sum on each node.

Second, the code generated for this example would be the same independent of the actual distribution chosen for the arrays as long as they remain identically aligned. The only change occurs in the descriptor of the distribution object, d. However, if the three vectors were individually distributed and not aligned, explicit communication would have to be inserted for each component operation.

4.5.2 Sparse Matrix Vector Multiply

This case study is based on a specific data parallel approach to a sparse matrix vector multiplication originally proposed in [31]. We outline the sequential algorithm, discuss the distribution and representation of the matrix in the memory of a Gilgamesh system, and subsequently formulate a pseudocode for a data parallel algorithm within our framework.

Sequential Algorithm

Consider the operation $S = A.B$, where $A(1:N, 1:M)$ is a sparse matrix with q nonzero elements, and $B(1:M)$ and $S(1:N)$ are vectors. The nonzero elements of A are enumerated using row-major order. An example for such a matrix has been already discussed in Section 4.2 and illustrated in Fig.8.

In the *Compressed Row Storage (CRS)* format, A is represented by three vectors, D, C, and R: (1) the *data vector*, $D(1:q)$, stores the sequence of nonzero elements of A, in the order of their enumeration; (2) the *column vector*, $C(1:q)$, contains in position k the column number, in A, of the k-th nonzero element in A; and (3) the *row vector*, $R(1:N+1)$, contains in position i the number of the first nonzero element of A in that row (if any); else the value of R(i+1). $R(N+1)$ is set to $q+1$. Based upon this representation, the core loop of the sequential algorithm can be formulated in Fortran as shown in Figure 10.

```
INTEGER :: C(q), R(N+1)
REAL :: D(q), B(M), S(N)
INTEGER :: I, J
 DO I = 1,M
    S(I)=0.0
    DO K = R(I), R(I+1)-1
       S(I) = S(I) + D(K)*B(C(K))
    ENDDO K
 ENDDO I
```

Figure 10: Sparse matrix vector multiply: core loop of sequential algorithm

Distributed Sparse Representation

The first step in developing a parallel version of the algorithm consists of defining a *distributed sparse representation* of A. This essentially combines a data distribution with a sparse format such as, in our case, CRS. The distributed sparse representation is then obtained by representing the submatrices constituting the distribution segments in the CRS format. Fig.11 extends the distribution of our matrix A as introduced in Fig.8 by the CRS representations of its distribution segments.

Macroserver-Based Parallel Algorithm

Based upon this representation we can derive a parallel algorithm for the sparse matrix vector product, as outlined in Figure 12. In order to keep this algorithm relatively simple, we focus on the local submatrix-subvector product and omit the actual partitioning algorithm as well as details such as dynamic array allocation and the computation of the final global sum.

The local matrix-vector product is computed in the method *mat_vec_loc*, which is activated as a separate thread, $T(u)$, in each node u. The distribution segments are given as $A_u = A(L1(u):U1(u), L2(u):U2(u))$, based on their global bounds, and $q(u)$ provides the number of nonzero elements in A_u. Furthermore, we assume that the components of the CRS representation for A_u are given by the array sections $D(u,1:q(u))$, $C(u,1:q(u))$, and $R(u,L1(u):U1(u)+1)$. Finally, each thread $T(u)$ stores its contribution to the partial sum in the temporary vector $TS(u,1:N)$.

The algorithm begins by creating the macroserver *my_sparse*, based on the class definition *sparse_template*. In the next step, the sparse matrix is generated and distributed, creating a distributed sparse CRS format. As a result of this step, the local representations $D(u,:)$, $C(u,:)$, and $R(u,:)$ as well as all the auxiliary data structures such as the arrays $L1, U1, L2, U2$, and q are set up. Once this is done, the NN threads $T(u)$ can be generated. They compute partial vectors which are stored in $TS(u,:)$. Finally, the $TS(u,:)$ are combined in a global sum to determine the final result vector, S. As mentioned in the previous example, any parallelism present in the algorithm at the thread level can be exploited locally on the node.

The vector B is distributed in some appropriate way which is not further specified here, and no communication involving B is included.

5 Related Work

Research experiments with semiconductor devices that merged both logic and static RAM cell blocks on the same chips have been conducted for more than a decade. Even earlier, simple processors and small blocks of SRAM could be found on control processors for embedded applications, and of course modern microprocessors

include high speed SRAM on-chip caches. Early projects include the J-Machine [11] and TERASYS [14] as well as EXECUBE [23], the first DRAM-based chip supporting a hypercube multiprocessor. But it was not until recently that industrial semiconductor fabrication processes made possible tightly coupled combinations of logic with DRAM cell blocks, bringing relatively large memory capacities to PIM design. A host of research projects has since been undertaken to explore this new design and application space. PIM is being pursued as a means of accelerating array processing in the Berkeley IRAM project [28], and for providing a smart memory in conventional systems in the FlexRAM and DIVA projects [21, 16, 17]. It has also been considered for the management of systems resources in a hybrid technology multithreaded architecture (HTMT) for ultra-scale computing [30]. In 1999, IBM announced the Blue Gene (BG/C) project [2, 3]. Similar to Gilgamesh Blue Gene is based on PIM nodes with multithreading technology; unlike Gilgamesh it does not support global virtual memory and uses a conventional approach towards communication. The Blue Gene project has already developed a software infrastructure for the system without access to real hardware, based on the ISA specification and an instruction-level simulator. Another recent development is a streaming supercomputer based on PIM technology currently developed at Stanford University [12]. Multithreading as well as a variant of parcels are also used in PIM-based developments at Notre Dame University [8]. Parcels have been also been employed in various forms by the DIVA project [16, 17]; they were first conceptualized for the HTMT Petaflops architecture project [30].

Macroservers have their roots in the *Opus* language [26] developed jointly at ICASE, NASA Langley Research Center, and the University of Vienna. Opus provides a hybrid programming model for expressing multidisciplinary applications focusing on shared abstractions (SDAs), which establish an object-based layer on top of HPF. Other languages contributing to the design of macroservers include Orca [5], Agora [7], Concurrent C++ [22], pC++ [33], and the actor model of computation [1]. Collections were introduced in [29]. Distribution and alignment as attributes of dense arrays were proposed in many languages emerging during the 1990s, including Kali, Vienna Fortran, Fortran D, HPF, and HPC++ [25, 9, 13, 20, 19]. The idea of user-defined distributions first originated in Kali [25] and was then put into a more concrete form in Vienna Fortran [9]. A more complete discussion of the generalized approach outlined in this paper for macroservers can be found in [36]. Work on the implementation of macroservers can draw in part on a rich body of experience with university prototypes as well as industrial software systems that were developed around the languages mentioned above, in particular implementations of HPF-like languages for distributed-memory systems (see, for example [10, 15, 6]). The last of these papers contains an extensive set of references to related compilation work.

6 Conclusions and Future Work

This paper presented salient design aspects of the Gilgamesh hardware architecture and its execution model. A base level MIND architecture is currently being prototyped using a specially designed testbed prototype board containing four high density FPGAs and 8 MB of SRAM to represent two MIND nodes and their on-chip interconnects. The parcel transport layer is provided by conventional Firewire interfaces. While the prototype board operates at one tenth the speed of what an actual chip would be capable of and is on a completely different fabrication density than an IC, its performance is at least a thousand times greater of a gate level cycle-by-cycle software simulator and its flexibility for rapid prototyping greatly exceeds that of the conventional IC design and fabrication cycle. This platform will allow early development of low level runtime system software to support the mechanisms and policies required to realize the dynamic adaptive operation of the system and management of virtual data and task objects. This will give us an accurate measure of time and space costs for implementing such fine-grain capabilities.

Current software activity focuses on a revised specification of macroserver functionality along the lines described in this paper and including the new set of features centered around distributed collections. We also develop a directive-based extension of Fortran 95 as a first user API, and analyze the associated compilation and runtime technology. While existing work, in particular for HPF-like languages provides a starting point, the generality of the macroserver approach implies the need for significant additional research and practical experiments in this area.

The study of PIM behavior must be supported – like that of any other parallel system – by a sophisticated set of software tools for performance analysis and high-level debugging. The light-weight thread mechanism provided by the Gilgamesh hardware suggests the deployment of persistent *monitor threads* for delivering performance information, checking program and system invariants, and performing behavior auditing for

detecting unusual program behavior. The feedback provided in this way can be also used for debugging, dynamic program tuning and fault analysis and recovery. In particular the last item represents an issue of central significance for future PIM-based PetaFlops systems: it will require an integrated hardware/software approach to address this critical problem in a manner which makes such systems viable.

The future role of PIM-based architectures will be decided in the context of other developments such as smart caches in conventional systems, the fast growth of memory system sizes, SMPs on a chip and the approach of the billion-transistor chip. It seems safe to say that the Gilgamesh architecture will make a significant contribution on the path to Petaflops computing and beyond.

Acknowledgement: The work described in this paper was partially supported by the National Aeronautics and Space Administration under NASA Contract No. NAS7-1407, Task Order No. 10652 (Gilgamesh Project), and the Special Research Program SFB011 "AURORA" of the Austrian Science Fund. The authors would like to thank Maciej Brodowicz, Mike Newell, and Edwin Upchurch for many fruitful discussions on this subject. We also thank the anonymous referees for their suggestions which have significantly influenced the final form of this paper.

References

[1] G.A.Agha. ACTORS: A Model of Concurrent Computation in Distributed Systems. *MIT Press*, 1986.

[2] G.S.Almasi,C.Cascaval,J.G.Castanos,M.Denneau,W.Donath, M.Eleftheriou, M.Giampapa, H.Ho, D.Lieber, J.E.Moreira, D.Newns, M.Snir, and H.S.Warren,Jr. Demonstrating the Scalability of a Molecular Dynamics Application on a Petaflop Computer. Research Report RC 21965, *IBM T.J. Watson Research Center*, Yorktown Heights, NY 10598, February 2001.

[3] G.S.Almasi,C.Cascaval,J.G.Castanos, D.Lieber, and J.E.Moreira. Developing System Software for Blue Gene. Research Report RC 21999, *IBM T.J. Watson Research Center*, Yorktown Heights, NY 10598, March 2001.

[4] G.R.Andrews. Concurrent Programming. Principles and Practice. *Benjamin/Cummings*, 1991.

[5] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[6] S.Benkner and H.P.Zima. Compiling High Performance Fortran for Distributed-Memory Architectures. In: Trystram,D.(Ed.): *Parallel Computing 25 (1999), Special Anniversary Issue*, pp.1785-1825.

[7] R. Bisiani, and A. Forin. Multilanguage Parallel Programming of Heterogeneous Machines *IEEE Transactions on Computers*, 37(8):930–945, August 1988.

[8] J.B.Brockman,P.M.Kogge,V.W.Freeh,S.K.Kuntz, and T.L.Sterling. Microservers: A New Memory Semantics for Massively Parallel Computing. *Proceedings ACM International Conference on Supercomputing (ICS'99)*, June 1999.

[9] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, Fall 1992.

[10] S.Chacrabarti,M.Gupta, and J.-D.Choi. Global Communication Analysis and Optimization. *Proc.ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'96*,pp.68-78,Philadelphia,PA, May 1996.

[11] W.J.Dally et al. The Message Driven Processor: A Multicomputer Processing Node With Efficient Mechanisms. *IEEE Micro*, April 1992,pp.23-28.

[12] W.Dally,P.Hanrahan, and R.Fedkiw. A Streaming Supercomputer. White Paper, *http://graphics.stanford.edu/sss*, November 2001.

[13] G.Fox,S.Hiranandani,K.Kennedy,C.Koelbel,U.Kremer,C.Tseng, and M. Wu. Fortran D language specification. *Department of Computer Science Rice COMP TR90079*, Rice University, March 1991.

[14] M.Gokhale,W.Holmes,and K.Iobst. Processing in Memory: The Terasys Massively Parallel PIM Array. *IEEE Computer 28(4)*,pp.23-31,1995.

[15] M.Gupta,E.Schonberg, and H.Srinivasan. A Unified Framework for Optimizing Communication in Data-Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*,Vol.7,No.7,pp.689-704, July 1996.

[16] M.Hall,J.Koller,P.Diniz,J.Chame,J.Draper, J.LaCoss, J.Granacki, J.Brockman, A.Srivastava, W.Athas, V.Freeh, J.Shin, and J.Park. Mapping Irregular Applications to DIVA, a PIM-Based Data Intensive Architecture. *Proceedings SC'99*, November 1999.

[17] J.Draper,J.Chame,M.Hall,C.Steele,T.Barrett,J.LaCoss,J.Granacki,J.Shin, C.Chen,C.W.Kang,I.Kim,and G.Daglikoca. The Architecture of the DIVA Processing-In-Memory Chip. *Proceedings ACM International Conference on Supercomputing (ICS'02)*, June 2002.

[18] R.H.Halstead,Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Transactions on Programming Languages and Systems (TOPLAS), 7(4),501-538*, October 1985.

[19] High Performance C++. WWW page *http://www.extreme.indiana.edu/hpc++/index.html*

[20] High Performance Fortran Forum. *High Performance Fortran Language Specification, Version 2.0*, January 1997.

[21] Y.Kang,W.Huang,S.-M.Yoo,D.Keen,Z.Ge,V.Lam,P.Pattnaik, and J.Torrellas. FlexRAM: Toward an Advanced Intelligent Memory System. *Proc.International Conf.on Computer Design (ICCD)*, Austin, Texas, October 1999.

[22] C.Kesselman. CC++. In: G.V.Wilson and P.Lu (Eds.): *Parallel Programming Using CC++*, Chapter 3, pp.91–130, The MIT Press, 1996.

[23] P.M.Kogge. The EXECUBE Approach to Massively Parallel Processing. In: *Proc.1994 Conference on Parallel Processing*, Chicago, August 1994.

[24] G. I. Marchuk. *Methods of Numerical Mathematics.* Springer-Verlag, 1975.

[25] P. Mehrotra and J. Van Rosendale. Programming distributed memory architectures using Kali. In A. Nicolau, D. Gelernter, T. Gross, and D. Padua, editors, *Advances in Languages and Compilers for Parallel Processing*, pages 364–384. Pitman/MIT-Press, 1991.

[26] P.Mehrotra,J.Van Rosendale, and H.P. Zima. Language Support for Multidisciplinary Applications. *IEEE Computational Science and Engineering* Vol.5,No.2,pp.64-75 (April-June 1998).

[27] P.Mehrotra,J.Van Rosendale, and H.P. Zima. High Performance Fortran: History, Status and Future. In: Zapata,E. and Padua,D.(Eds.): *Parallel Computing, Special Issue on Languages and Compilers*, Vol.24,No.3-4,pp.325–354 (1998)

[28] D.PattersonT.Anderson,N.Cardwell,R.Fromm,K.Keeton,C.Kozyrakis,R.Tomas, and K.Yelick. A Case for Intelligent DRAM: IRAM. *IEEE Micro*, pp.33-44, April 1997.

[29] J.M.Sipelstein,G.E.Blelloch. Collection-Oriented Languages. *Proceedings of the IEEE*,Vol.79,No.4,pp.504-523, April 1991.

[30] T.Sterling and L.Bergman. A Design Analysis of a Hybrid Technology Multithreaded Architecture for Petaflops Scale Computation. *Proceedings ACM International Conference on Supercomputing (ICS'99)*, June 1999.

[31] M.Ujaldon,E.L.Zapata,B.M.Chapman,and H.P.Zima. Vienna Fortran/HPF Extensions for Sparse and Irregular Problems and Their Compilation. *IEEE Transactions on Parallel and Distributed Systems*, Vol.8, No.10, pp.1068-1083 (October 1997).

[32] T.Von Eicken,D.E.Culler,S.C.Goldstein,and K.E.Schauser. Active Messages: a Mechanism for Integrated Communication and Computation. *Proc.19th Int.Symposium on Computer Architecture*,Gold Coast, Australia, ACM Press (1992).

[33] S.X.Yang,D.Gannon,P.Beckman,J.Gotwals, and N.Sundaresan. pC++. In: G.V.Wilson and P.Lu (Eds.): *Parallel Programming Using CC++*, Chapter 13, pp.507–545, The MIT Press, 1996.

[34] H.Zima and T.Sterling. Macroservers. An Execution Model for DRAM Processor-In-Memory Arrays. *Technical Report CACR-182, Center for Advanced Computing Research, California Institute of Technology*, Pasadena, CA, February 2000. Also: *Technical Report TR 00-01, Institute for Software Science, University of Vienna, February 2000.*

[35] H.Zima and T.Sterling. Macroservers: An Object-Based Programming and Execution Model for Processor-in-Memory Arrays. *Proc.International Symposium on High Performance Computing (ISHPC2K)*, Tokyo, Japan, October 2000.

[36] H.Zima. Data Distribution Specification for High Performance Computing. *Journal of Universal Computer Science (JUCS), Special Issue on Formal Aspects of Software Engineering - J.UCS Special Issue in Honor of Professor Peter Lucas*, Vol.7, No. 8 (2001), pp.736-753.

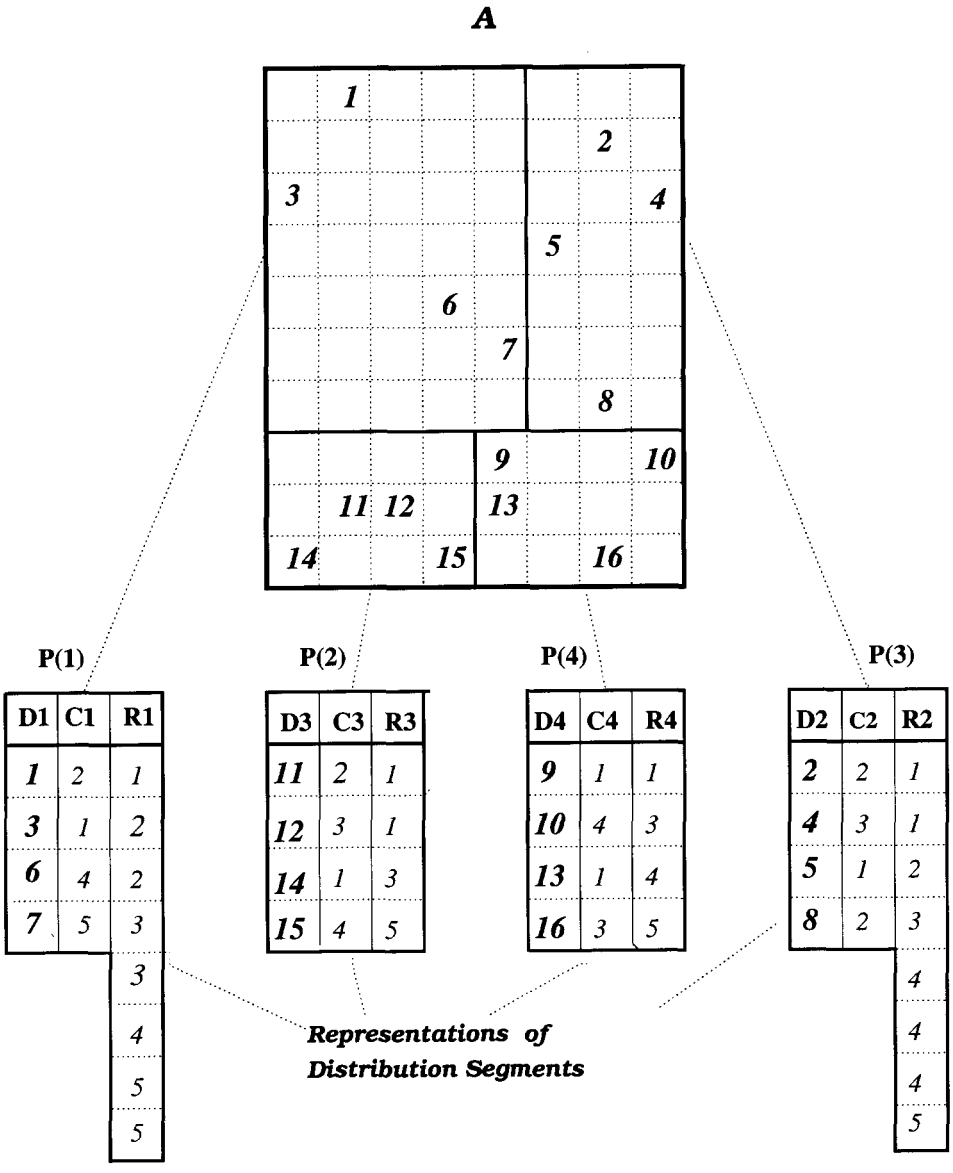

Figure 11: Distributed representation of sparse matrix A

```
INTEGER NN = number_of_nodes()    ! number of nodes available for this application
NODES P(NN)                       ! abstract node array
MACROSERVER CLASS sparse_template
  INTEGER :: u
  REAL, SPARSE (CRS(L1,U1,L2,U2,D,C,R,q,...),P) :: A(N,M)
  INTEGER :: L1(NN), U1(NN), L2(NN), U2(NN), q(NN)
  REAL :: D(NN,:), C(NN,:), R(NN,:)
  REAL, DISTRIBUTE (...,:) TO (P) :: TS(NN,:)
  REAL, DISTRIBUTE (...) TO (P) :: B(M)
  REAL :: S(N)
  FUTURE :: T(NN)

  CONTAINS

  METHOD generate_and_partition()
    ! Generates matrix, determines MRD partition, and sets up the distributed data structures and auxiliary data
  END METHOD generate_and_partition

  METHOD mat_vec_loc(u,D,C,R)
  REAL :: D(q(u))
  INTEGER :: C(q(u)), R(L1(u):U1(u)+1)
  INTEGER :: I, K
   DO I = L1(u),U1(u)
      TS(u,1:N) = 0.0
      DO K = R(I), R(I+1)-1
         TS(u,I) = TS(u,I) + D(K)*B(C(K)+L2(u))
      ENDDO K
   ENDDO I
  END METHOD mat_vec_loc

  METHOD global_sum()
    ! Performs global reductions to compute final result vector, stored in S, from the temporary vectors TS(u,:).
    ! This is a parallel algorithm that uses the future variables bound to the threads generated
    ! in the matrix_vector routine for synchronization.
  END METHOD global_sum

  METHOD matrix_vector()
       ! Create on each node a thread executing a parameterized version of mat_vec_loc. Provide
       ! arguments that point to the local distribution segments.
     FORALL THREADS (u=1:NN, ON HOME (A(L1(u):U1(u),L2(u):U2(u)))
              T(u) = SPAWN (my_sparse%mat_vec_loc,u,D(u,:),C(u,:),R(u,:))
  END METHOD matrix_vector
END MACROSERVER CLASS sparse_template

! Main program
MACROSERVER (sparse_template) my_sparse = CREATE (sparse_template)
CALL my_sparse%generate_and_partition()
CALL my_sparse%matrix_vector()
CALL my_sparse%global_sum()
```

Figure 12: Parallel sparse matrix-vector multiply using macroservers